Analytic Functions

/*Analytic Functions

After the database server has completed all of the steps necessary to evaluate a query, including joining, filtering, grouping, and sorting, the result set is complete and ready to be returned to the caller. If your result set contains sales data, perhaps you might want to generate rankings for salespeople or regions, or calculate percentage differences between one time period and another. If you are generating results for a financial report, perhaps you would like to calculate subtotals for each report section, and a grand total for the final section. Using analytic functions, you can do all of these things and more.*/

/*Data Windows

write a query that generates monthly sales totals for a may to august 0f 2005 period.*/

```
SELECT
    QUARTER(payment_date) quarter,
    MONTHNAME(payment_date) month_name,
    SUM(amount) monthly_sales
FROM
    payment
WHERE
    YEAR(payment_date) = 2005
GROUP BY QUARTER(payment_date) , MONTHNAME(payment_date);
```

```
SELECT
    QUARTER(payment_date) quarter,
    MONTHNAME(payment_date) month_name,
    SUM(amount) monthly_sales
FROM
    payment
WHERE
    YEAR(payment_date) = 2005
GROUP BY 1,2;


SELECT
      quarter(payment_date) AS quarter,
      monthname(payment_date) AS month_name,
      sum(amount) AS monthly_sales,
      max(sum(amount)) over () max_overall_sales,
      max(sum(amount)) over (partition by quarter(payment_date)) AS
max_qrtr_sales
FROM
      payment
WHERE
      year(payment_date) = 2005
GROUP BY quarter(payment_date), monthname(payment_date);
```

/*To accommodate this type of analysis, analytic functions include the ability to group rows into windows, which effectively partition the data for use by the analytic function without changing the overall result set. **Windows** are defined using the over clause combined with an optional partition by subclause.

In the previous query, both analytic functions include an over clause, but the first one is empty, indicating that the window should include the entire result set, whereas the second one specifies that the window should include only rows within the same quarter.

Localized Sorting*/

```
SELECT
      quarter(payment_date) AS quarter,
      monthname(payment_date) AS month_nm,
      sum(amount) AS monthly_sales,
      rank() over (order by sum(amount) desc) AS sales_rank
FROM
      payment
WHERE
      year(payment_date) = 2005
GROUP BY quarter(payment_date), monthname(payment_date)
ORDER BY 1, month(payment_date);
```

```
SELECT
      quarter(payment_date) quarter,
      monthname(payment_date) month_nm,
      sum(amount) monthly_sales,
      rank() over (partition by quarter(payment_date)
      order by sum(amount) desc) qtr_sales_rank
FROM
      payment
WHERE
      year(payment_date) = 2005
GROUP BY quarter(payment_date), monthname(payment_date)
ORDER BY 1, month(payment_date);
```

/*Ranking

There are multiple ranking functions available in the SQL standard, with each one taking a different approach to how ties are handled:

row_number -

Returns a unique number for each row, with rankings arbitrarily assigned in case of a tie

rank -

Returns the same ranking in case of a tie, with gaps in the rankings

dense_rank -

Returns the same ranking in case of a tie, with no gaps in the rankings

*/

```
SELECT
    customer_id, COUNT(*) AS number_of_rentals
FROM
    rental
GROUP BY customer_id
ORDER BY 2 DESC;


SELECT
     customer_id,
     count(*) num_rentals,
     row_number() over (order by count(*) desc) row_number_rnk,
     rank() over (order by count(*) desc) rank_rnk,
     dense_rank() over (order by count(*) desc) dense_rank_rnk
FROM
     rental
GROUP BY
     customer_id
ORDER BY 2 desc;
```

/*Generating Multiple Rankings*/

```
SELECT
    customer_id,
    MONTHNAME(rental_date) rental_month,
    COUNT(*) num_rentals
FROM
    rental
GROUP BY customer_id , MONTHNAME(rental_date)
ORDER BY 2 , 3 DESC;

SELECT
     customer_id,
     monthname(rental_date) rental_month,
     count(*) num_rentals,
     rank() over (partition by monthname(rental_date)
     order by count(*) desc) rank_rnk
FROM
     rental
GROUP BY
     customer_id, monthname(rental_date)
ORDER BY 2, 3 desc;
```

```
SELECT
      customer_id,
      rental_month,
    num_rentals,
      rank_rnk ranking
FROM
      (SELECT
            customer_id,
            monthname(rental_date) rental_month,
            count(*) num_rentals,
            rank() over (partition by monthname(rental_date)
            order by count(*) desc) rank_rnk
      FROM
            rental
      GROUP BY
            customer_id, monthname(rental_date)
      ) cust_rankings
WHERE rank_rnk <= 5
ORDER BY rental_month, num_rentals desc, rank_rnk;


/*Since analytic functions can be used only in the SELECT clause, you
will often need to nest queries if you need to do any filtering or
grouping based on the results from the analytic function.*/
```

/*Reporting Functions

```
Query that generates monthly and grand totals for all payments of $10
or higher*/


SELECT
      monthname(payment_date) payment_month,
      amount,
      sum(amount)
      over (partition by monthname(payment_date)) monthly_total,
      sum(amount) over () grand_total
FROM
      payment
WHERE
      amount >= 10
ORDER BY 1;
```

/*Calculatoion using grand_total column*/

```
SELECT
      monthname(payment_date) payment_month,
      sum(amount) month_total,
      round(sum(amount)  /  sum(sum(amount))  over  ()  *  100,  2)
pct_of_total
FROM payment
GROUP BY monthname(payment_date);
```

/*Window Frames

Data windows for analytic functions are defined using the partition by clause, which allows you to group rows by common values.*/

```
SELECT
    SUM(amount)
FROM
    payment;


SELECT
      yearweek(payment_date) payment_week,
      sum(amount) week_total,
      sum(sum(amount))
      over (order by yearweek(payment_date)
            rows unbounded preceding) rolling_sum
FROM
      payment
GROUP BY
      yearweek(payment_date)
ORDER BY 1;
```

/*rolling avg*/

```
SELECT
      yearweek(payment_date) payment_week,
      sum(amount) week_total,
      avg(sum(amount))
      over (order by yearweek(payment_date)
            rows between 1 preceding and 1 following) rolling_3wk_avg
FROM
      payment
GROUP BY
      yearweek(payment_date)
ORDER BY 1;
```

```
/*The rolling_3wk_avg column defines a data window consisting of the current row, the prior row, and the next row. The data window will therefore consist of three rows, except for the first and last rows, which will have a data window consisting of just two rows (since there is no prior row for the first row and no next row for the last row).*/
SELECT
      date(payment_date), sum(amount),
      avg(sum(amount)) over (order by date(payment_date)
            range between interval 3 day preceding
                  and interval 3 day following) 7_day_avg
FROM
      payment
WHERE
      payment_date BETWEEN '2005-07-01' AND '2005-09-01'
GROUP BY
      date(payment_date)
ORDER BY 1;
```

/*Lag and Lead

Along with computing sums and averages over a data window, another common reporting task involves comparing values from one row to another.

For example, if you are generating monthly sales totals, you may be asked to create a column showing the percentage difference from the prior month, which will require a way to retrieve the monthly sales total from the previous row. This can be accomplished using the lag function, which will retrieve a column value from a prior row in the result set, or the lead function, which will retrieve a column value from a following row.*/

```
SELECT
      yearweek(payment_date) payment_week,
      sum(amount) week_total,
      lag(sum(amount), 1)
            over (order by yearweek(payment_date)) prev_wk_tot,
      lead(sum(amount), 1)
            over (order by yearweek(payment_date)) next_wk_tot
FROM
      payment
GROUP BY
      yearweek(payment_date)
ORDER BY 1;
```

/*percentage difference using lag function*/

```
SELECT
      yearweek(payment_date) payment_week,
      sum(amount) week_total,
      round((sum(amount) - lag(sum(amount), 1)
            over (order by yearweek(payment_date)))
            / lag(sum(amount), 1)
                  over (order by yearweek(payment_date))
            * 100, 1) pct_diff
FROM
      payment
GROUP BY
      yearweek(payment_date)
ORDER BY 1;
```

/*Column Value Concatenation

The group_concat function is used to pivot a set of column values into a single delimited

string, which is a handy way to denormalize your result set for generating XML or

JSON documents.*/

```
SELECT
    f.title,
    GROUP_CONCAT(a.last_name
        ORDER BY a.last_name
        SEPARATOR ', ') actors
FROM
    actor a
        INNER JOIN
    film_actor fa ON a.actor_id = fa.actor_id
        INNER JOIN
    film f ON fa.film_id = f.film_id
GROUP BY f.title
HAVING COUNT(*) = 3;
```

```
/*Exercise - For all exercises in this section, use the following data
set from the Sales_Fact table: Sales_Fact

+---------+----------+-----------+
| year_no | month_no | tot_sales |
+---------+----------+-----------+
| 2019    | 1        | 19228     |
| 2019    | 2        | 18554     |
| 2019    | 3        | 17325     |
| 2019    | 4        | 13221     |
| 2019    | 5        | 9964      |
| 2019    | 6        | 12658     |
| 2019    | 7        | 14233     |
| 2019    | 8        | 17342     |
| 2019    | 9        | 16853     |
| 2019    | 10       | 17121     |
| 2019    | 11       | 19095     |
| 2019    | 12       | 21436     |
| 2020    | 1        | 20347     |
| 2020    | 2        | 17434     |
| 2020    | 3        | 16225     |
| 2020    | 4        | 13853     |
| 2020    | 5        | 14589     |
| 2020    | 6        | 13248     |
| 2020    | 7        | 8728      |
| 2020    | 8        | 9378      |
| 2020    | 9        | 11467     |
| 2020    | 10       | 13842     |
| 2020    | 11       | 15742     |
| 2020    | 12       | 18636     |
+---------+----------+-----------+*/
```

/*Exercise - 1

Write a query that retrieves every row from Sales_Fact, and add a column to generate a ranking based on the tot_sales column values. The highest value should receive a ranking of 1, and the lowest a ranking of 24.*/

```
SELECT
      year_no,
      month_no,
    tot_sales,
      rank() over (order by tot_sales desc) sales_rank
FROM sales_fact;
```

/*Exercise 2

Modify the query from the previous exercise to generate two sets of rankings from 1 to 12, one for 2019 data and one for 2020.*/

```
SELECT
      year_no,
    month_no,
    tot_sales,
      rank() over (partition by year_no order by tot_sales desc) sales_rank
FROM sales_fact;
```

/*Exercise - 3

Write a query that retrieves all 2020 data, and include a column that will contain the tot_sales value from the previous month.*/

```
SELECT
     year_no,
    month_no,
    tot_sales,
     lag(tot_sales) over (order by month_no) prev_month_sales
FROM
     sales_fact
WHERE
     year_no = 2020;
```